# Minimize Mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
  - Disadvantage: separate object for each value
- If mutable, keep state-space small, well-defined
  - Make clear when it's legal to call which method

Bad: `Date`, `Calendar`

Good: `TimerTask`

## Subclass Only Where It Makes Sense

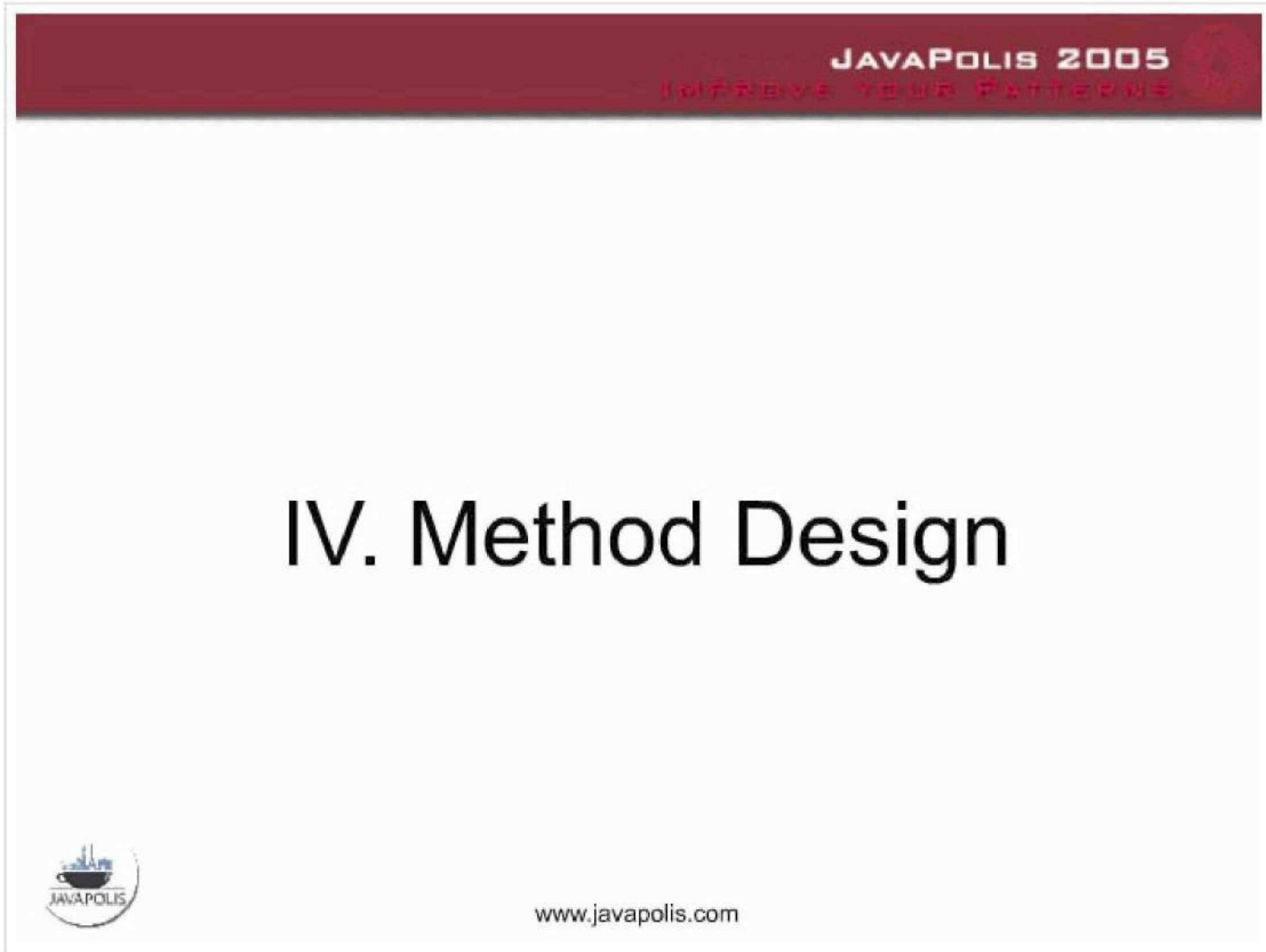
- Subclassing implies substitutability (Liskov)
  - Subclass only when is-a relationship exists
  - Otherwise, use composition
- Public classes should not subclass other public classes for ease of implementation

Bad:
```
Properties extends Hashtable
Stack extends Vector
```

Good:
```
Set extends Collection
```

# Design and Document for Inheritance or Else Prohibit it

- Inheritance violates encapsulation (Snyder, '86)
  - Subclass sensitive to implementation details of superclass
- If you allow subclassing, document *self-use*
  - How do methods use one another?
- Conservative policy: all concrete classes final
- Bad: Many concrete classes in J2SE libraries
- Good: `AbstractSet`, `AbstractMap`



# Don't Make the Client Do Anything the Module Could Do

- Reduce need for boilerplate code
  - Generally done via cut-and-paste
  - Ugly, annoying, and error-prone

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// DOM code to write an XML document to a specified output stream.
static final void writeDoc(Document doc, OutputStream out)throws IOException{
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
  doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out));
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```

# Don't Violate *Principle of Least Astonishment*

- User of API should not be surprised by behavior
  - It's worth extra implementation effort
  - It's even worth reduced performance

```
public class Thread implements Runnable {
  // Tests whether current thread has been interrupted.
  // Clears the interrupted status of current thread.
  public static boolean interrupted();
}
```

# Fail Fast–Report Errors as Soon as Possible After They Occur

- Compile time is best - static typing, generics
- At runtime, first bad method invocation is best
  - Method should be failure-atomic

```
// A Properties instance maps strings to strings
public class Properties extends Hashtable {
   public Object put(Object key, Object value);

   // Throws ClassCastException if this properties
   // contains any keys or values that are not strings
   public void save(OutputStream out, String comments);
}
```

# Provide Programmatic Access to All Data Available in String Form

- Otherwise, clients will parse strings
  - Painful for clients
  - Worse, turns string format into de facto API

```
public class Throwable {
  public void printStackTrace(PrintStream s);
  public StackTraceElement[] getStackTrace(); // Since 1.4
}
public final class StackTraceElement {
  public String getFileName();
  public int getLineNumber();
  public String getClassName();
  public String getMethodName();
  public boolean isNativeMethod();
}
```

# Overload With Care

- Avoid *ambiguous overloadings*
  - Multiple overloadings applicable to same actuals
  - Conservative: no two with same number of args
- Just because you can doesn't mean you should
  - Often better to use a different name
- If you must provide ambiguous overloadings, ensure same behavior for same arguments

```
public TreeSet(Collection c); // Ignores order
public TreeSet(SortedSet s);  // Respects order
```

# Use Appropriate Parameter and Return Types

- Favor interface types over classes for input
  - Provides flexibility, performance
- Use most specific possible input parameter type
  - Moves error from runtime to compile time
- Don't use string if a better type exists
  - Strings are cumbersome, error-prone, and slow
- Don't use floating point for monetary values
  - Binary floating point causes inexact results!
- Use `double` (64 bits) rather than `float` (32 bits)
  - Precision loss is real, performance loss negligible

# Use Consistent Parameter Ordering Across Methods

- Especially important if parameter types identical

```
#include <string.h>
char *strcpy (char *dest, char *src);
void bcopy   (void *src,  void *dst, int n);
```

`java.util.Collections` – first parameter always collection to be modified or queried

`java.util.concurrent` – time always specified as `long delay, TimeUnit unit`

# Avoid Long Parameter Lists

- Three or fewer parameters is ideal
  - More and users will have to refer to docs
- Long lists of identically typed params harmful
  - Programmers transpose parameters by mistake
  - Programs still compile, run, but misbehave!
- Two techniques for shortening parameter lists
  - Break up method
  - Create helper class to hold parameters

```
// Eleven parameters including four consecutive ints
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
      DWORD dwStyle, int x, int y, int nWidth, int nHeight,
      HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
      LPVOID lpParam);
```

# Avoid Return Values that Demand Exceptional Processing

- Return zero-length array or empty collection, not **null**

```
package java.awt.image;
public interface BufferedImageOp {
  // Returns the rendering hints for this operation,
  // or null if no hints have been set.
  public RenderingHints getRenderingHints();
}
```

# Throw Exceptions to Indicate Exceptional Conditions

- Don't force client to use exceptions for control flow

```
private byte[] a = new byte[BUF_SIZE];
void processBuffer (ByteBuffer buf) {
  try {
    while (true) {
      buf.get(a);
      processBytes(tmp, BUF_SIZE);
    }
  } catch (BufferUnderflowException e) {
    int remaining = buf.remaining();
    buf.get(a, 0, remaining);
    processBytes(bufArray, remaining);
  }
}
```

- Conversely, don't fail silently

```
ThreadGroup.enumerate(Thread[] list)
```

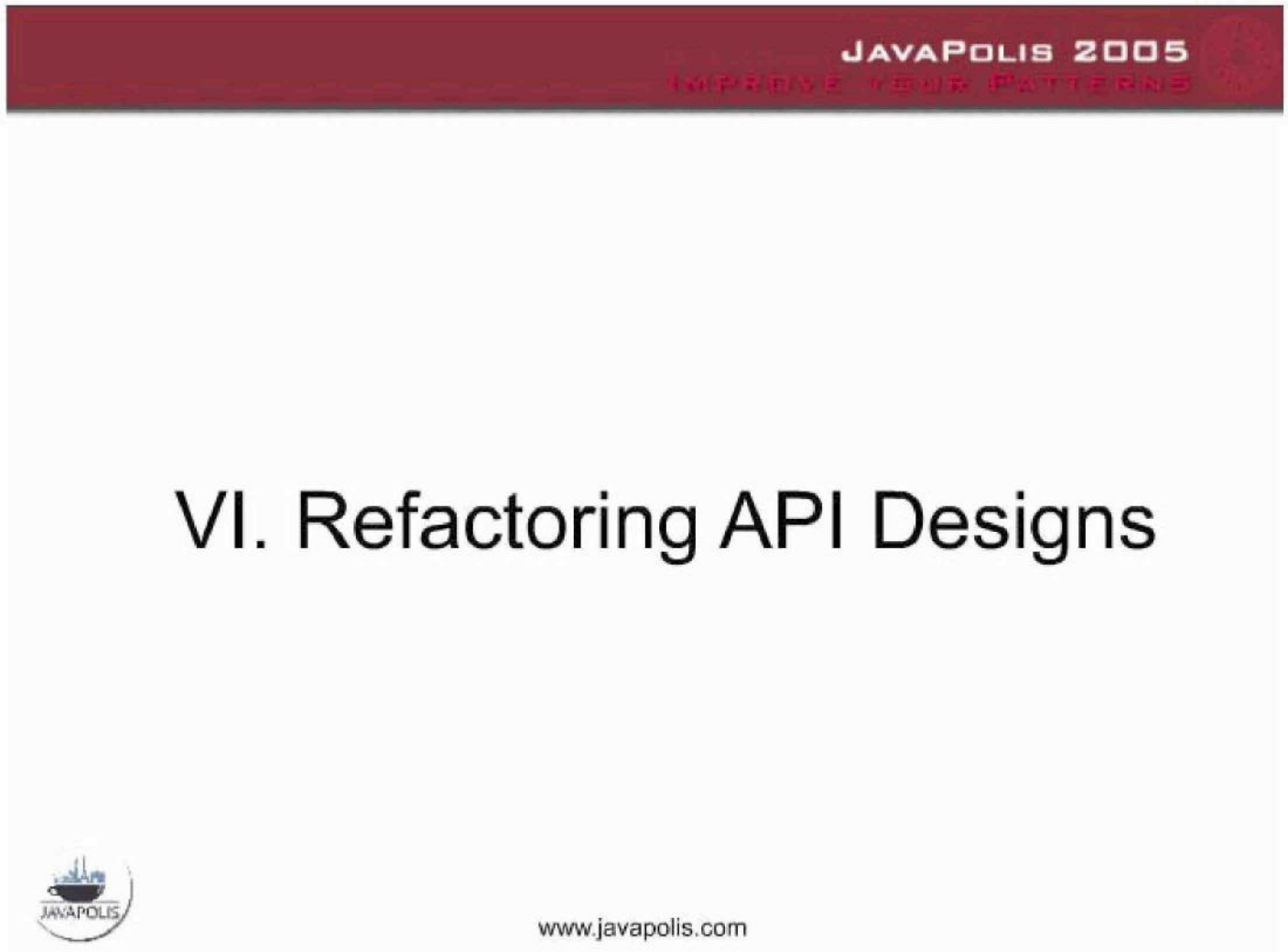
# Favor Unchecked Exceptions

- Checked – client must take recovery action
- Unchecked – programming error
- Overuse of checked exceptions causes boilerplate

```
try {
    Foo f = (Foo) super.clone();
    ....
} catch (CloneNotSupportedException e) {
    // This can't happen, since we're Cloneable
    throw new AssertionError();
}
```

## Include Failure-Capture Information in Exceptions

- Allows diagnosis and repair or recovery
- For unchecked exceptions, message suffices
- For checked exceptions, provide accessors



# 1. Sublist Operations in Vector

```
public class Vector {
    public int indexOf(Object elem, int index);
    public int lastIndexOf(Object elem, int index);
    ...
}
```

- Not very powerful - supports only search
- Hard too use without documentation

# Sublist Operations Refactored

```
public interface List {
    List subList(int fromIndex, int toIndex);
    ...
}
```

- Extremely powerful - supports *all* operations
- Use of interface reduces conceptual weight
  - High power-to-weight ratio
- Easy to use without documentation

# 2. Thread-Local Variables

```
// Broken - inappropriate use of String as capability.
// Keys constitute a shared global namespace.
public class ThreadLocal {
    private ThreadLocal() { } // Non-instantiable

    // Sets current thread's value for named variable.
    public static void set(String key, Object value);

    // Returns current thread's value for named variable.
    public static Object get(String key);
}
```

# Thread-Local Variables Refactored (1)

```
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    public static class Key { Key() { } }

    // Generates a unique, unforgeable key
    public static Key getKey() { return new Key(); }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

- Works, but requires boilerplate code to use

```
static ThreadLocal.Key serialNumberKey = ThreadLocal.getKey();
ThreadLocal.set(serialNumberKey, nextSerialNumber());
System.out.println(ThreadLocal.get(serialNumberKey));
```

# Thread-Local Variables Refactored (2)

```
public class ThreadLocal {
    public ThreadLocal() { }
    public void set(Object value);
    public Object get();
}
```

- Removes clutter from API and client code

```
static ThreadLocal serialNumber = new ThreadLocal();
serialNumber.set(nextSerialNumber());
System.out.println(serialNumber.get());
```

# Conclusion

- API design is a noble and rewarding craft
  - Improves the lot of programmers, end-users, companies
- This talk covered some heuristics of the craft
  - Don't adhere to them slavishly, but...
  - Don't violate them without good reason
- API design is tough
  - Not a solitary activity
  - Perfection is unachievable, but try anyway